“It began with blotting-paper,” the Knight answered with a groan.
“That wouldn’t be very nice, I’m afraid—”
“Not very nice alone,” he interrupted, quite eagerly: “but you’ve no idea what a difference it makes, mixing it with other things—such as gunpowder and sealing wax.”

Through the Looking-Glass

As we have now seen, Common Lisp has a range and power exceeded by no other popular language. These very features, however, put the novice programmer in a quandary. How best should a particular function be used? What optional keyword arguments are there that might simplify the code? Is there a more appropriate function anyway? Even the experienced programmer can sometimes feel like a child in a toy supermarket—simply spoilt for choice.
I have, to maintain the analogy, deliberately kept the upper shelves from view so far, occasionally lifting the covers to reveal briefly a gleaming fire engine or sophisticated train set. In truth, it is not the function of this book to do more than this, but I feel that it is at least desirable to expand some topics a little, revise some misconceptions that I have introduced (deliberately!) and point the way towards further investigations with your system manual and other texts.

Functions, Macros and Special Forms

First, I must come clean on the subject of Lisp functions. I have called all Lisp forms that evaluate with zero or more arguments functions. In the strictly mathematical sense (provided there are no side effects) this is true enough. But in practice there are considerations that require certain 'functions' to be handled differently. Recall that in Chapter 12 I discussed briefly the way in which a stack is brought into existence for every function call and then destroyed on exit. This is clearly time consuming, so that if a function is to be called frequently, the delays caused may be unacceptable. An alternative is to expand the call into the code that would be called if it were a function. The code is now in-line and there is no need for sophisticated communication mechanisms. This is what is meant by a macro. Examples of macros (that I have already misdescribed as functions) are push and pop.

There is a third category of apparent 'functions'. These are the special forms. As a rule, they are control constructs with some unusual syntactic features. Examples are if and let.

In fact, when the evaluator is given a list, it examines its head and tries to match it to a special form, a macro and a function in that order.

defmacro

We have managed fine without this knowledge so far and it does not appear to affect the applications programmer much. However, it is possible to define your own macros and, in doing so, save significant processing time. For example, I defined head and tail as functions in Chapter 2 to relieve us of the burden of having to interpret car and cdr. A better alternative would have been to define them as macros, since each reference to head would then be replaced by car rather than initiating an expensive function call. Here's how it's done:

```lisp
(defmacro head (x)
  (list 'car x))

(defmacro tail (x)
  (list 'cdr x))
```

The form of defmacro is similar to that of defun in that it begins with the macro name and a list of arguments (here just x). The body of the macro differs, however, in that it is not a set of actions to take, but a piece of replacement text.

Thus we are saying here, "On encountering a list consisting of head with one